Multithreading

Introduction

Although increasing performance by using ILP has the great advantage that it is reasonably transparent to the programmer, it can be quite limited or difficult to exploit in some applications. Since attempts to cover long memory stalls with ILP have limited effectiveness, it is natural to ask whether other forms of parallelism in an application could be used to hide memory delays.

Multithreading allows multiple threads to share the functional unit of a single processor in an overlapping fashion. In contrast, a more general method to exploit thread-level parallelism is with a **multiprocessor**, which has multiple independent threads operating at once and in parallel.

Multithreading does not duplicate the entire processor as a multiprocessor does, however. Instead, multithreading shares most of the processor core among a set of threads, duplicating only private state—such as the registers and the program counter.

Duplicating the per-thread state of a processor core means creating a separate register file, a separate program counter, and a separate page table for each thread. The memory itself can be shared through virtual memory mechanisms. In addition, the hardware must support the ability to change to a different thread relatively quickly: a thread switch should be **much more efficient than a process switch**, which typically requires hundreds to thousands of processor cycles.

For multithreaded hardware to achieve performance improvements, a program **must contain multiple threads** that could execute in concurrent fashion. These are identified by either a compiler—typically from a language with parallelism constructs—or by the programmer.

There are three main approaches to multithreading, which we will describe below.

Fine-grained multithreading

Fine-grained multithreading switches between threads on each clock, causing the execution of instructions from multiple threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that may be stalled.

Fine-grained multithreading can hide the throughput losses that arise from short and long stalls, since instructions from other threads can still be executed.

The primary disadvantage of this technique is that it slows down the execution of an individual thread: **it trades an increase in multithreaded throughput for a loss in single-threaded performance**.

Coarse-grained multithreading

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading: it switches threads on costly stalls only, such as level two or three cache misses.

In coarse-grained multithreading, the need for very quick thread switching is not as strong as it was in fine-grained multithreading. Furthermore, a lower frequency of thread switching will greatly reduce the slowdown of a single thread.

Nevertheless, coarse-grained multithreading is limited in its ability to **overcome throughput losses**, especially from **short stalls**: because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs the pipeline will see a bubble before the new thread begins executing. Because of this startup overhead, this kind of multithreading is far more useful for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous multithreading

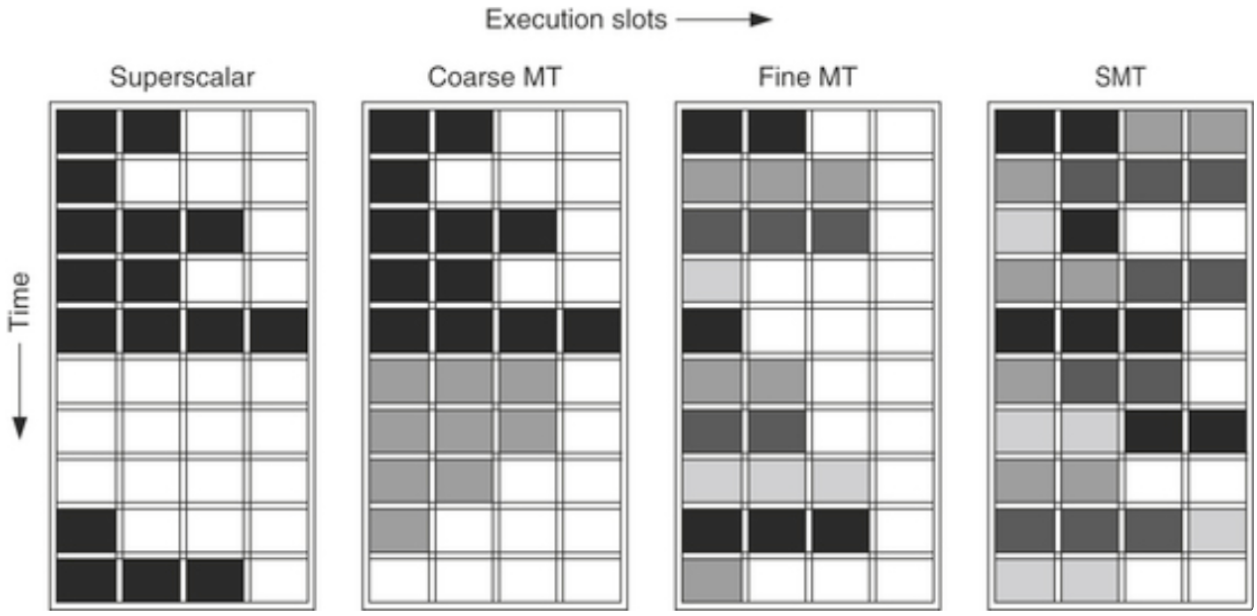
The most common implementation of multithreading is known as **simultaneous multithreading**. Simultaneous multithreading is a variation on fine-grained multithreading that arises naturally when the latter is implemented on top of a multiple-issue, dynamically scheduled processor.

As with other types of multithreading, this technique uses thread-level parallelism in order to hide long latency events in a processor, thus increasing the usage of the functional units.

The key insight in simultaneous multithreading is that register renaming and dynamic scheduling allow multiple instructions from **independent threads** to be executed **without regard to the dependencies among them**; the resolution of the dependencies can be handled by the dynamic scheduling capability.

Comparing multithreading techniques

The image below illustrates the differences between the above mentioned types of multithreading and a non-multithreaded, superscalar processor:



In the superscalar with no multithreading, the use of issue slots is limited by lack of ILP and because of the length of L2 and L3 cache misses, much of the processor can be left idle.

In the coarse-grained multithreaded superscalar, long stalls are partially hidden by switching to another thread. This reduces the number of completely idle clock cycles, but there are likely to be some remaining, since the switching occurs only then a stall is detected.

In the fine-grained case, the interleaving of threads eliminates fully empty slots and longer latency operations can be hidden. However, because instruction issue and execution are connected, a thread can only issue as many instructions as are ready. With a narrow issue width, this is not a problem, which is why this technique works perfectly with single issue processors.

Finally, if one implements fine-grained on top of a multiple-issue, dynamically scheduled processor, simultaneous multithreading—or SMT—is obtained. In all SMT implementations, all issues come from one thread, although instructions from different threads can initiate execution in the same cycle, using the dynamically scheduling hardware to determine which instructions are ready. SMT uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support it, including a large virtual register set: multithreading can be built on top of an out-of-order processor by adding a per-

thread renaming table, keeping separate program counters, and providing the capability for instructions from multiple threads to commit.